CS335A: Milestone 1

Sawan H N (210952) Shrilakshmi S K (211012) Yashas D (211199)

March 3, 2024

1 Required Environment

For the environment setup, it is essential to have Flex and Bison installed. On Linux systems, these tools can be installed using the commands `sudo apt install flex` for Flex and `sudo apt install bison` for Bison, respectively. Additionally, the system must have the `g++` compiler available for use. To visualize the `.dot` file produced when the parser file is executed, the Graphviz tool is utilized.

2 Instructions to run the code:

Go to directory `cs335-project-39` and run the following commands in terminal:

```
cd milestone/src
make clean
make
cd ..
./run.sh -i <input.py> -o <output.pdf>
```

All the tests are in `tests` directory and all the outputs are in `outputs` directory. Note that `make` might fail sometimes on Windows. Please retry if it does. On Linux, if any file denies permission, please execute `sudo chmod +x <filename>` for that file.

3 Options

How to use: `./run.sh` [options]
Options:
`-i Specify input (.py) file`
`-o Specify output (.pdf) file`
`-v Enable verbose mode`
`-h Show help`

4 Lexical Analyzer

We have written rules for all the tokens necessary to implement Python 3.8 grammar. The following list encompasses key implementation details of lexer.

4.1 Implicit Line Joining

To enhance the width of cases covered and facilitate certain syntax structures, our system supports implicit line joining under all conditions. Some key conditions include:

- Nested function calls, where one function call is used as an argument within another function call.
- Function calls placed within lists.
- The declaration and use of multi-dimensional lists.

To manage this functionality, we employ a state-based approach, transitioning to a distinct state upon encountering an opening bracket (`(` or `[`) and reverting to the previous state when a closing bracket (`)` or `]`) is encountered. This method allows us to differentiate between parentheses and square brackets, ensuring support for the scenarios outlined above.

For state management, we utilize a stack that records state transitions, enabling the system to return accurately to prior states once bracket pairs are closed. This stack mechanism tracks the depth and context of nested structures.

4.2 Indentation

We use a separate stack dedicated to managing indentation levels. This stack helps us generate the appropriate indent and dedent tokens, crucial for understanding the structure of the code, especially in languages where indentation is syntactically significant. A special state is designated for handling situations where multiple dedent tokens need to be passed to the parser, ensuring the parser receives the correct cues to interpret the code structure effectively.

4.3 Other Cases

We have also handled explicit line joining, blank lines and comments especially within multiline lists.

5 Changes to Grammar

We have implemented Python 3.8 grammar for this milestone. To remove shift-reduce conflicts, reduce the number of states and make the grammar more comprehensible, we have further made the following changes.

- Included operators and operands in same production rule for expressions.
- Removed epsilon productions causing shift-reduce conflicts.
- Removed optional semicolon and comma at the end of a sequence of small statements and list of arguments or tests respectively. This has been done because official python shows warning when semicolon is placed at the end of sequence of small statements. This change does not affect future stages of compilation in any manner.
- Removed right recursion in some productions to reduce the number of non-terminals.
- Introduced few non-terminals to incorporate the regular expressions given in standard grammar to Bison.

6 Constructing AST

We first built the parse tree and further made the following changes to construct an AST.

- Removed all non-terminals with single child from the graph. We connected the children of child of non-terminal with single child directly to the said non-terminal. This vastly improves comprehensibility of AST.
- For expressions, the operators are made parents and the operands are its children.
- Labelled the edges with relevant attributes. For example, the type of variable such as `int` or `str` or `bool` has an incoming edge with the label `"type"`. Another example is in if-statements, the condition expression has an incoming edge with label `"if"` and the body of if condition has an incoming edge with label `"then"`. To improve comprehensibility, the enveloping brackets such as `()` and `[]` are also labeled on edges. Similarly, edges are labelled in function definitions, class definitions, for statements and while statements.
- As a result of labelling edges, we have removed some redundant terminals from AST such as `IF, ELSE, ELIF, FOR, IN, WHILE, COMMA, SEMICOLON, COLON, DEF, CLASS, OPEN_SQUARE_BRACKET, CLOSE_SQUARE_BRACKET, OPEN_ROUND_BRACKET` and `CLOSE_ROUND_BRACKET`. These terminals are easily understood from the edge labels.

7 Additional Features

We have handled all the features as required by milestone. Additionally we have supported the following features.

- Multi-dimensional lists
- Comments under all cases including in-between multi-line lists, multi-line function calls and definitions.
- Explicit type casting
- Escape sequences in strings

- Assert statements
- Any number of spaces are supported for indentation (similar to official Python). It need not necessarily be 2 spaces.

Some features like nonlocal statements, import statements, del keyword, with statements, pass keyword and raising exceptions can be easily included in the parser. But these are excluded due to uncertainty in implementation of these features in future stages of compilation.

8 Reporting Errors

The lexer reports indentation error and invalid character error with line number and exits the program. We have used `%define parser.error verbose` in parser to report the errors where the input does not obey the grammar's productions. It also reports expected token and line number where error is encountered.

CS335A: Milestone 2

Sawan H N (210952) Shrilakshmi S K (211012) Yashas D (211199)

April 1, 2024

1 Required Environment

For the environment setup, it is essential to have Flex and Bison installed. On Linux systems, these tools can be installed using the commands `sudo apt install flex` for Flex and `sudo apt install bison` for Bison, respectively. Additionally, the system must have the `g++` compiler available for use.

2 Instructions to run the code:

Go to directory `cs335-project-39` and run the following commands in terminal:

```
cd milestone2
./make.sh
./run.sh -i <input.py>
```

All the `.csv` files of symbol tables and 3AC code `3AC.txt` will be outputted in `milestone2` directory. Note that `make` might fail sometimes on Windows. Please retry if it does. On Linux, if any file denies permission, please execute `sudo chmod +x <filename>` for that file.

3 Options

How to use: `./run.sh` [options]
Options:

```
-i Specify input (.py) file
-v Enable verbose mode
-h Show help
```

4 Changes to Grammar

We have introduced some constraints on official Python 3.8 grammar. LHS of the expression statements cannot be constants. Further, we have divided the expression statements into initialization expression statements and non-initialization expression statements.
Following are the valid LHS for initialization expression statements:

- `NAME`
- `NAME DOT NAME` (where the former NAME can only be "self")

Following are the valid LHS for non-initialization expression statements:

- `NAME`
- `NAME DOT NAME`
- `NAME [test]`
- `NAME DOT NAME [test]`

Further we have following expression statements with no RHS.

- `test`
- `NAME (arglist)`
- `NAME DOT NAME (arglist)`

We have handled all the above cases.

5 Symbol Table

We have created different symbol tables for different scopes. Class, Function and Global are the scopes. We do not have different scopes for control statement blocks, as it was unnecessary. We store token, type, offset, scope, initialization status, number of parameters and parameter number in symbol table.

If a class is child of another class, we copy all the symbol table entries of parent class and proceed with populating the table further.

6 Supported Features

- Class inheritance: Child classes can access parent class's attributes and methods. While accessing methods, type of `self` argument is checked.
- Method call by both objects and classes

```
1  class Student:
2      def __init__(self, name : str, age : int):
3          self.name : str = name
4          self.age : int = age
5      def get_grade(self, num : int) -> int:
6          grade : int = self.age - num
7          return grade
8
9  student1 : Student = Student("Shril", 20)
10 grade_obj : int = student1.get_grade(13)  # method
   ↪ call by object without 'self' argument
11 grade_class : int = Student.get_grade(student1, 13)
   ↪ # method call by class with 'self' argument as
   ↪  object
```

- Implicit typecasting of `int <---> bool` and `int <---> float`. This typecasting is supported for expression evaluations, assignments, parameters type matching and return value from functions. For expression evaluations, if types of two operands differ, we have always typecasted `int` to `float` and `bool` to `int`. During assignment, they might

get typecasted back depending upon the type hint given for LHS. We have also supported typecasting of child class type into parent class type.

- We have implemented the `range()` function in our 3AC code.
- We have supported type checking in all possible scenarios: All the assignment expressions stated in previous section, return values, parameter types including `self` parameter while also supporting child typecast to parent.
- Global variables have been supported. Our version of global variable means that they cannot be declared again in any scope.
- `break` and `continue` statements are supported in for and while loops.

7 3AC Code Generation

7.1 Expression Statements

We followed an analogy similar to AST construction in creating 3AC code for expression statements. We created a new temporary variable if there was an operation in the production of the non-terminal, else we passed on the temporary variable to higher non-terminals in the parse tree.

```
1 def main():
2     x : int = 2 * 3 + 4 / 5 ** 9
```

```
1  main:
2      beginfunc
3      push %rbp
4      %rbp = %rsp
5      t0 = 2
6      t1 = 3
7      t2 = t0 * t1
8      t3 = 4
9      t4 = 5
10     t5 = 9
11     t6 = t4 ** t5
12     t7 = t3 / t6
```

```
    t8 = t2 + t7
    x = t8
    sub %rsp 4
    -4(%rbp) = x
    add %rsp 4
    pop %rbp
    ret
    endfunc
```

7.2 Control Flow Statements

For "if-elif-else" statements, we have used a stack to store the exit labels of elif statements. For "for statements", we have internally implemented the `range()` function.

7.2.1 If Statement

```
def main():
    a : int = 2
    if a > 3:
        a = 1
```

```
main:
    beginfunc
    push %rbp
    %rbp = %rsp
    t0 = 2
    a = t0
    sub %rsp 4
    -4(%rbp) = a
    t1 = 3
    t2 = a > t1
    Ifz t2 Goto L0
    t3 = 1
    a = t3
    -4(%rbp) = a
    Goto L1
L0:
```

```
L1:
     add %rsp 4
     pop %rbp
     ret
     endfunc
```

7.2.2 While Statement

```
def main():
     a : int = 2
     while a < 1:
          a -= 1
```

```
main:
     beginfunc
     push %rbp
     %rbp = %rsp
     t0 = 2
     a = t0
     sub %rsp 4
     -4(%rbp) = a
L0:
     t1 = 1
     t2 = a < t1
     Ifz t2 Goto L1
     t3 = 1
     t4 = a - t3
     a = t4
     -4(%rbp) = a
     Goto L0
L1:
     add %rsp 4
     pop %rbp
     ret
     endfunc
```

7.2.3 For Statements

```
def main():
    a : int = 3
    i : int
    b : int = 0
    for i in range(a):
        b = i
```

```
main:
    beginfunc
    push %rbp
    %rbp = %rsp
    t0 = 3
    a = t0
    sub %rsp 4
    -4(%rbp) = a
    t1 = 0
    b = t1
    sub %rsp 4
    -12(%rbp) = b
    t2 = 0
    i = t2
    Goto L1
L0:
    t5 = i + 1
    i = t5
L1:
    t6 = i < a
    Ifz t6 Goto L2
    b = i
    -12(%rbp) = b
    Goto L0
L2:
    add %rsp 12
    pop %rbp
    ret
    endfunc
```

7.3 Lists

Since list size cannot be pre-determined, especially when it is a parameter in a function, we have pointers for lists. List size, then becomes 8 bytes - the size of a pointer. We first call `allocmem` and allocate the memory for list and initialize the elements of the list one by one. As lists are mutable, this approach is useful to modify individual elements of list.

```
def main():
    a : int = 3
    i : int
    b : int = 0
    for i in range(a):
        b = i
```

```
main:
    beginfunc
    push %rbp
    %rbp = %rsp
    t0 = 2
    t1 = 3
    t2 = 5
    t3 = 6
    t4 = 16
    pushparam t4
    push ret_addr
    sub %rsp 4
    call allocmem 1
    add %rsp 4
    t5 = %rax
    *(t5 + 0) = t0
    *(t5 + 4) = t1
    *(t5 + 8) = t2
    *(t5 + 12) = t3
    a = t5
    sub %rsp 8
    -8(%rbp) = a
    t6 = 0
    t7 = t6 * 4
    t8 = 1
    *(a + t7) = t8
```

```
27      add %rsp 8
28      pop %rbp
29      ret
30      endfunc
```

7.4 Objects

Similar to list, to construct objects, we allocate the size of object in heap using allocmem, when constructor function is called. We then assign the values of object attributes based on offsets stored in symbol table.

```
1  class Student:
2      def __init__(self, name : str, age : int):
3          self.name : str = name
4          self.age : int = age
5
6  def main():
7      s : Student = Student("Shril", 19)
8      s.age = 20
```

```
1  Student.__init__:
2      beginfunc
3      push %rbp
4      %rbp = %rsp
5      self = 16(%rbp)
6      sub %rsp 8
7      -8(%rbp) = self
8      name = 24(%rbp)
9      sub %rsp 8
10     -16(%rbp) = name
11     age = 32(%rbp)
12     sub %rsp 4
13     -20(%rbp) = age
14     *(self + 0) = name
15     *(self + 8) = age
16     add %rsp 20
17     pop %rbp
18     ret
19     endfunc
```

```
main:
    beginfunc
    push %rbp
    %rbp = %rsp
    t0 = "Shril"
    t1 = 19
    t2 = 12
    pushparam t2
    sub %rsp 4
    call allocmem 1
    add %rsp 4
    t3 = %rax
    pushparam t1
    pushparam t0
    pushparam t3
    push ret_addr
    sub %rsp 20
    call Student.__init__ 3
    add %rsp 20
    s = t3
    sub %rsp 8
    -8(%rbp) = s
    t4 = 20
    *(s + 8) = t4
    add %rsp 8
    pop %rbp
    ret
    endfunc
```

7.5 Typecasting

We have used a method `cast_to_type` to typecast in 3AC code.

```
def add(a : int, b : float) -> float:
    c : int = a + b
    return c

def main():
    res : int = add(2.3, 3)
```

```
add:
    .........
    -8(%rbp) = b
    t0 = cast_to_float a
    t1 = t0 + b
    t2 = cast_to_int t1
    c = t2
    sub %rsp 4
    -12(%rbp) = c
    t3 = cast_to_float c
    ..........
main:
    ..........
    t4 = 2.3
    t5 = 3
    t6 = cast_to_int t4
    t7 = cast_to_float t5
    pushparam t7
    pushparam t6
    .........
```

8 Runtime Support

8.1 After function is called

When a function is called, we push `%rbp` on the stack. Then we are accessing the parameters of callee function (beginning from `16(%rbp)`) from the stack and storing them as local variables again. We are also pushing other local variables to the stack. We modify `%rsp` according to their widths. Whenever we modify these local variables, we are also modifying the value in the stack. For pointers such as objects and lists, this is not necessary as their value modifications gets reflected in stack too when we change them in the function. To return, we are storing the value of return value in `%rax` register. Then we pop all the local variables, we adjust `%rsp` accordingly. We then pop old `%rbp` and change `%rbp`. We lastly use instruction `ret` which pops the return address from the stack and `%rip` continues its execution from that address. **All of these register manipulations are reflected in our 3AC code.**

```
add:
    beginfunc
    push %rbp
    %rbp = %rsp
    a = 16(%rbp)
    sub %rsp 4
    -4(%rbp) = a
    b = 20(%rbp)
    sub %rsp 4
    -8(%rbp) = b
    ...........
```

8.2 Calling a function

To call a function, we first push the parameters of callee function on the stack using `pushparam`. We modify `%rsp` according to their widths. We then push return address. We then call the function using `call func_name num_params` line in 3AC code. **All of these register manipulations are reflected in our 3AC code.**

```
main:
    ...........
    t4 = 2
    t5 = 3.2
    pushparam t5
    pushparam t4
    push ret_addr
    sub %rsp 8
    call add 2
    add %rsp 8
    ..........
```

8.3 Returning from a function

If the return value exists, then it will be stored in `%rax` register. We assign this value to a temporary variable. Then we assign this temporary variable to the return variable. **All of these register manipulations are reflected in our 3AC code.**

```
main:
     ...........
     t6 = %rax
     res = t6
     ..........
```

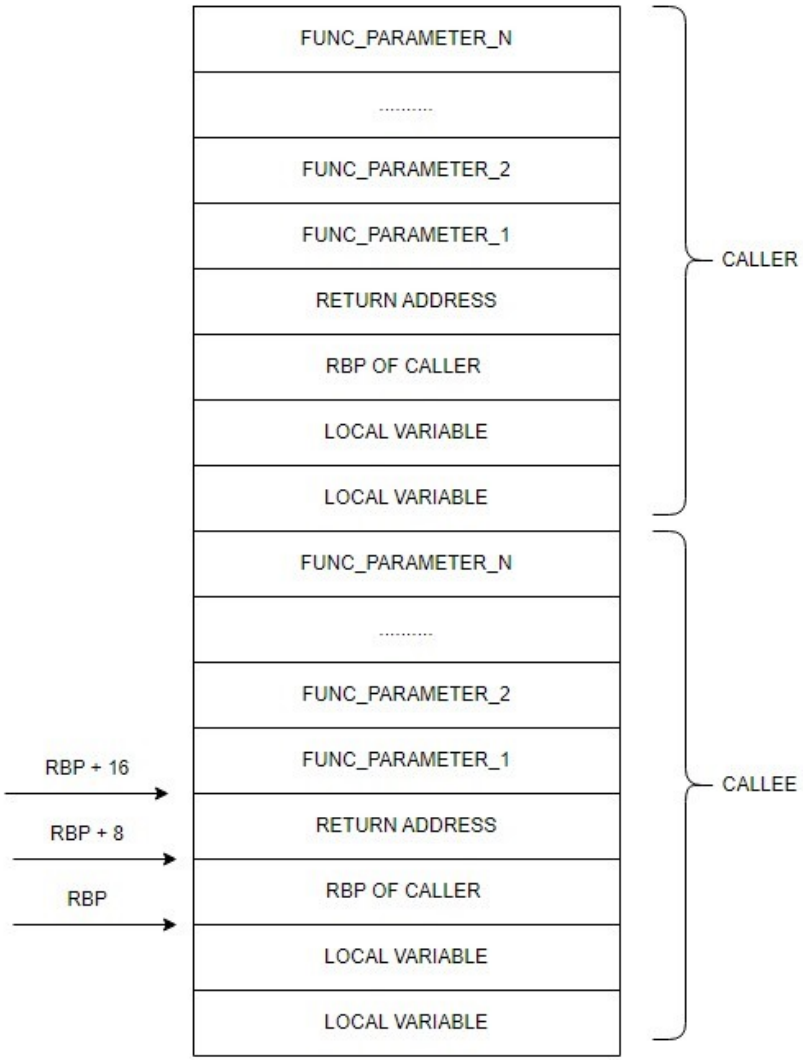

Figure 1: Stack for Runtime Support

9 Error Handling

This is the non-exhaustive list of errors reported with **separate** error messages. Error messages are reported at **160+** places in the parser code with line numbers.

- Redeclaration of any symbol table entry
- Redeclaration of global variable in any scope
- Global variable cannot be declared (`x : int = 5`) before global declaration (`global x`)
- Return type error based on the function's return type hint and function's return value. We have also handled the case of None return type
- Type hints for parameters not given
- Invalid assignments for LHS and RHS types
- `self` cannot be used as variable name. It is reserved for instances of classes.
- Number of arguments passed in functions and methods (including print(), len() and range())
- Types of arguments passed in functions and methods (including print(), len() and range()) including support for typecasting
- Undeclared function/method error
- Undeclared variable/attribute error
- **Uninitialised variable/attribute error** - If a variable/attribute is used in RHS without initialization of some value beforehand, error is reported
- Constructor call parameter errors
- Undeclared object error
- Undeclared class error

- Invalid operation error based on the operator and types of operands
- Attribute reinitialization error
- Initialization of self attribute in non-constructor function error
- List index type must be int
- Index error, for indexing non-list types
- Index error, for indexing lists with higher dimensions
- Iterating variable in for loop must be int
- Base / parent class does not exist error

10 Assumptions

- `__init__()` function / constructor does not have return type mentioned
- `self` does not have type mentioned in parameters
- No stray code in classes
- All the statements are enclosed in functions except for

```
1 if __name__ == "__main__":
2     main()
```

- Parent class must be defined before child class, else an error will be thrown
- Functions are defined before calling in any another function

11 Contribution

- Sawan - 33.3%
- Shrilakshmi - 33.3%
- Yashas - 33.3%

CS335A: Milestone 3

Sawan H N (210952) Shrilakshmi S K (211012) Yashas D (211199)

April 20, 2024

1 Required Environment

For the environment setup, it is essential to have Flex and Bison installed. On Linux systems, these tools can be installed using the commands `sudo apt install flex` for Flex and `sudo apt install bison` for Bison, respectively. Additionally, the system must have the `g++` compiler available for use.

2 Instructions to run the code:

Go to directory `cs335-project-39` and run the following commands in terminal:

```
cd milestone3
./make.sh
./run.sh -i <input.py>
./script.sh
```

The 3AC code `3AC.txt` and x86 assembly code `x86.s` will be outputted in `milestone3` directory. Note that `make` might fail sometimes on Windows. Please retry if it does. On Linux, if any file denies permission, please execute `sudo chmod 777 <filename>` for that file.

3 Options

How to use: `./run.sh` [options]
Options:
`-i Specify input (.py) file`
`-v Enable verbose mode`
`-h Show help`

4 Supported Features

- Class inheritance: Child classes can access parent class's attributes and methods. While accessing methods, type of `self` argument is checked.
- Method call by both objects and classes

```
class Student:
    def __init__(self, name : str, age : int):
        self.name : str = name
        self.age : int = age
    def get_grade(self, num : int) -> int:
        grade : int = self.age - num
        return grade

student1 : Student = Student("Shril", 20)
grade_obj : int = student1.get_grade(13)  # method call by object without 'self' argument
grade_class : int = Student.get_grade(student1, 13) # method call by class with 'self' argument as object
```

- Implicit typecasting of `int <---> bool`. This typecasting is supported for expression evaluations, assignments, parameters type matching and return value from functions. For expression evaluations, if types of two operands differ, we have always typecasted `int` to `float` and `bool` to `int`. During assignment, they might get typecasted back depending upon the type hint given for LHS. We have also supported typecasting of child class type into parent class type.

- We have implemented the `range()` and `len()` function internally. To support `len()`, we have stored the length of the list as the first element of the allocated memory.
- We have supported type checking in all possible scenarios: All the assignment expressions stated in previous section, return values, parameter types including `self` parameter while also supporting child typecast to parent.
- `break` and `continue` statements are supported in for and while loops.
- We have supported list indexing of an object attribute of list type in the same expression via `obj.attr[idx]`.
- String comparison is fully supported across all relational operators (`==, !=, <=, >=, <, >`)
- We have used `print@PLT`, `malloc@PLT`, `strcmp@PLT` in our code. For this purpose, we have aligned the stack pointer to offset of multiple of 16 before calling any function.

5 Modifications and Manual Changes

We have internally implemented some methods in x86 code from 3AC code to support stack alignment, string comparison, typecasting methods, and `len` function.

There are no manual changes required for generated x86 code. It can directly be run via GAS.

6 Error Handling

This is the non-exhaustive list of errors reported with **separate** error messages. Error messages are reported at **160+** places in the parser code with line numbers.

- Redeclaration of any symbol table entry
- Redeclaration of global variable in any scope

- Global variable cannot be declared (`x : int = 5`) before global declaration (`global x`)
- Return type error based on the function's return type hint and function's return value. We have also handled the case of None return type
- Type hints for parameters not given
- Invalid assignments for LHS and RHS types
- `self` cannot be used as variable name. It is reserved for instances of classes.
- Number of arguments passed in functions and methods (including print(), len() and range())
- Types of arguments passed in functions and methods (including print(), len() and range()) including support for typecasting
- Undeclared function/method error
- Undeclared variable/attribute error
- **Uninitialised variable/attribute error** - If a variable/attribute is used in RHS without initialization of some value beforehand, error is reported
- Constructor call parameter errors
- Undeclared object error
- Undeclared class error
- Invalid operation error based on the operator and types of operands
- Attribute reinitialization error
- Initialization of self attribute in non-constructor function error
- List index type must be int
- Index error, for indexing non-list types

- Index error, for indexing lists with higher dimensions
- Iterating variable in for loop must be int
- Base / parent class does not exist error

7 Assumptions

- The `__init__()` function (constructor) does not have a return type specified.
- The `self` parameter does not have a type specified in the parameters.
- There is no stray code outside in class definitions.
- If a class has no parent, then it is defined as `class Class_Name:` and not `class Class_Name():`.
- All statements are enclosed in functions, except for the following block:

```
1     if __name__ == "__main__":
2         main()
```

- The parent class must be defined before any child class; otherwise, an error will occur.
- Functions are defined before they are called within any other function.
- Nested functions and nested classes are not included in the input program.
- Nested function calls are not included in the input program.
- Floating point inputs are not considered.

8 Appendix : x86 Assembly Code Outputs

8.1 Expression Statements

We followed an analogy similar to AST construction in creating 3AC code for expression statements. We created a new temporary variable if there was an operation in the production of the non-terminal, else we passed on the temporary variable to higher non-terminals in the parse tree.

```
def main():
    x : int = 2 * 3 + 4 / 5 ** 9
```

```
        .section    .rodata
.LC0:
        .string "%ld\n"
.LC1:
        .string "%s\n"
.LC2:
        .string "__main__"
        .globl  main
        .text

# beginfunc main
main:
        pushq   %rbp
        movq    %rsp,   %rbp
        pushq   %rbx
        pushq   %rdi
        pushq   %rsi
        pushq   %r12
        pushq   %r13
        pushq   %r14
        pushq   %r15
        sub     $88,    %rsp

# #t0 = 2
        movq    $2, -64(%rbp)

# #t1 = 3
        movq    $3, -72(%rbp)
```

```
29
30 # #t2 = #t0 * #t1
31         movq    -64(%rbp),  %rdx
32         imulq   -72(%rbp),  %rdx
33         movq    %rdx,   -80(%rbp)
34
35 # #t3 = 4
36         movq    $4, -88(%rbp)
37
38 # #t4 = 5
39         movq    $5, -96(%rbp)
40
41 # #t5 = 9
42         movq    $9, -104(%rbp)
43
44 # #t6 = #t4 ** #t5
45         movq    $0, -112(%rbp)
46         movq    $1, -120(%rbp)
47         jmp     2f
48 1:
49         movq    -112(%rbp), %rdx
50         addq    $1, %rdx
51         movq    %rdx,   -112(%rbp)
52 2:
53         movq    -112(%rbp), %rdx
54         movq    -104(%rbp), %rcx
55         cmp     %rdx,   %rcx
56         je      3f
57         movq    -120(%rbp), %rdx
58         imulq   -96(%rbp),  %rdx
59         movq    %rdx,   -120(%rbp)
60         jmp     1b
61 3:
62
63 # #t8 = #t3 / #t6
64         movq    -88(%rbp),  %rax
65         cqto
66         movq    -120(%rbp), %rbx
67         idiv    %rbx
68         movq    %rax,   %rdx
```

```
69          movq    %rdx,   -128(%rbp)
70
71  # #t9 = #t2 + #t8
72          movq    -80(%rbp),  %rdx
73          addq    -128(%rbp), %rdx
74          movq    %rdx,   -136(%rbp)
75
76  # x = #t9
77          movq    -136(%rbp), %rdx
78          movq    %rdx,   -144(%rbp)
79
80  # end main
81          movq    $60,    %rax
82          xor     %rdi,   %rdi
83          syscall
84
85  # #t10 = "__main__"
86          leaq    .LC2(%rip), %rdx
87          movq    %rdx,   -64(%rbp)
88
89  # #t11 = __name__ == #t10
90          movq    (%rbp), %rdx
91          movq    -64(%rbp),  %rcx
92          movq    %rdx,   %rdi
93          movq    %rcx,   %rsi
94          call    strcmp@PLT
95          movsx   %eax,   %rax
96          cmp     $0, %rax
97          jne     1f
98          movq    $1, %rdx
99          jmp     2f
100 1:
101         movq    $0, %rdx
102 2:
103         movq    %rdx,   -80(%rbp)
104
105 # Ifz #t11 Goto .L0
106         movq    -80(%rbp),  %rdx
107         cmp     $0, %rdx
108         je      .L0
```

```
109
110 # aligning stack
111         pushq   $0
112
113 # funccall main
114         pushq   %rax
115         pushq   %rcx
116         pushq   %rdx
117         pushq   %r8
118         pushq   %r9
119         pushq   %r10
120         pushq   %r11
121
122 # call main 0
123         call    main
124         add     $0, %rsp
125         popq    %r11
126         popq    %r10
127         popq    %r9
128         popq    %r8
129         popq    %rdx
130         popq    %rcx
131         popq    %rax
132         add     $8, %rsp
133
134 # Goto .L1
135         jmp     .L1
136 .L0:
137 .L1:
```

8.2 Control Flow Statements

For "if-elif-else" statements, we have used a stack to store the exit labels of elif statements. For "for statements", we have internally implemented the `range()` function.

8.2.1 If Statement

```
1 def main():
```

```
    a : int = 2
    if a > 3:
        a = 1
```

```
        .section    .rodata
.LC0:
        .string "%ld\n"
.LC1:
        .string "%s\n"
.LC2:
        .string "__main__"
        .globl  main
        .text

# beginfunc main
main:
        pushq   %rbp
        movq    %rsp,   %rbp
        pushq   %rbx
        pushq   %rdi
        pushq   %rsi
        pushq   %r12
        pushq   %r13
        pushq   %r14
        pushq   %r15
        sub     $40,    %rsp

# #t0 = 2
        movq    $2, -64(%rbp)

# a = #t0
        movq    -64(%rbp),  %rdx
        movq    %rdx,   -72(%rbp)

# #t1 = 3
        movq    $3, -80(%rbp)

# #t2 = a > #t1
        movq    -72(%rbp),  %rdx
        movq    -80(%rbp),  %rcx
```

```
37          cmp     %rdx,   %rcx
38          jl      1f
39          movq    $0, %rdx
40          jmp     2f
41  1:
42          movq    $1, %rdx
43  2:
44          movq    %rdx,   -88(%rbp)
45
46  # Ifz #t2 Goto .L0
47          movq    -88(%rbp),  %rdx
48          cmp     $0, %rdx
49          je      .L0
50
51  # #t3 = 1
52          movq    $1, -96(%rbp)
53
54  # a = #t3
55          movq    -96(%rbp),  %rdx
56          movq    %rdx,   -72(%rbp)
57
58  # Goto .L1
59          jmp     .L1
60  .L0:
61  .L1:
62
63  # end main
64          movq    $60,    %rax
65          xor     %rdi,   %rdi
66          syscall
67
68  # #t4 = "__main__"
69          leaq    .LC2(%rip), %rdx
70          movq    %rdx,   -64(%rbp)
71
72  # #t5 = __name__ == #t4
73          movq    (%rbp), %rdx
74          movq    -64(%rbp),  %rcx
75          movq    %rdx,   %rdi
76          movq    %rcx,   %rsi
```

```
77          call    strcmp@PLT
78          movsx   %eax,   %rax
79          cmp     $0, %rax
80          jne     1f
81          movq    $1, %rdx
82          jmp     2f
83  1:
84          movq    $0, %rdx
85  2:
86          movq    %rdx,   -80(%rbp)
87
88  # Ifz #t5 Goto .L2
89          movq    -80(%rbp),  %rdx
90          cmp     $0, %rdx
91          je      .L2
92
93  # aligning stack
94          pushq   $0
95
96  # funccall main
97          pushq   %rax
98          pushq   %rcx
99          pushq   %rdx
100         pushq   %r8
101         pushq   %r9
102         pushq   %r10
103         pushq   %r11
104
105 # call main 0
106         call    main
107         add     $0, %rsp
108         popq    %r11
109         popq    %r10
110         popq    %r9
111         popq    %r8
112         popq    %rdx
113         popq    %rcx
114         popq    %rax
115         add     $8, %rsp
116
```

```
# Goto .L3
        jmp     .L3
.L2:
.L3:
```

8.2.2 While Statement

```
def main():
    a : int = 2
    while a < 1:
        a -= 1
```

```
        .section    .rodata
.LC0:
        .string "%ld\n"
.LC1:
        .string "%s\n"
.LC2:
        .string "__main__"
        .globl  main
        .text

# beginfunc main
main:
        pushq   %rbp
        movq    %rsp,   %rbp
        pushq   %rbx
        pushq   %rdi
        pushq   %rsi
        pushq   %r12
        pushq   %r13
        pushq   %r14
        pushq   %r15
        sub     $40,    %rsp

# #t0 = 2
        movq    $2, -64(%rbp)

# a = #t0
```

```
28          movq    -64(%rbp),  %rdx
29          movq    %rdx,   -72(%rbp)
30  .L0:
31
32  # #t1 = 1
33          movq    $1, -80(%rbp)
34
35  # #t2 = a < #t1
36          movq    -72(%rbp),  %rdx
37          movq    -80(%rbp),  %rcx
38          cmp     %rdx,   %rcx
39          jg      1f
40          movq    $0, %rdx
41          jmp     2f
42  1:
43          movq    $1, %rdx
44  2:
45          movq    %rdx,   -88(%rbp)
46
47  # Ifz #t2 Goto .L1
48          movq    -88(%rbp),  %rdx
49          cmp     $0, %rdx
50          je      .L1
51
52  # #t3 = 1
53          movq    $1, -96(%rbp)
54
55  # #t4 = a - #t3
56          movq    -72(%rbp),  %rdx
57          movq    -96(%rbp),  %rcx
58          subq    %rcx,   %rdx
59          movq    %rdx,   -72(%rbp)
60
61  # Goto .L0
62          jmp .L0
63  .L1:
64
65  # end main
66          movq    $60,    %rax
67          xor     %rdi,   %rdi
```

```
68          syscall
69
70  # #t5 = "__main__"
71          leaq    .LC2(%rip), %rdx
72          movq    %rdx,   -64(%rbp)
73
74  # #t6 = __name__ == #t5
75          movq    (%rbp), %rdx
76          movq    -64(%rbp),  %rcx
77          movq    %rdx,   %rdi
78          movq    %rcx,   %rsi
79          call    strcmp@PLT
80          movsx   %eax,   %rax
81          cmp     $0, %rax
82          jne     1f
83          movq    $1, %rdx
84          jmp     2f
85  1:
86          movq    $0, %rdx
87  2:
88          movq    %rdx,   -80(%rbp)
89
90  # Ifz #t6 Goto .L2
91          movq    -80(%rbp),  %rdx
92          cmp     $0, %rdx
93          je      .L2
94
95  # aligning stack
96          pushq   $0
97
98  # funccall main
99          pushq   %rax
100         pushq   %rcx
101         pushq   %rdx
102         pushq   %r8
103         pushq   %r9
104         pushq   %r10
105         pushq   %r11
106
107 # call main 0
```

```
        call    main
        add     $0, %rsp
        popq    %r11
        popq    %r10
        popq    %r9
        popq    %r8
        popq    %rdx
        popq    %rcx
        popq    %rax
        add     $8, %rsp

# Goto .L3
        jmp     .L3
.L2:
.L3:
```

8.2.3 For Statements

```
def main():
    a : int = 3
    i : int
    b : int = 0
    for i in range(a):
        b = i
```

```
        .section    .rodata
.LC0:
        .string "%ld\n"
.LC1:
        .string "%s\n"
.LC2:
        .string "__main__"
        .globl  main
        .text

# beginfunc main
main:
        pushq   %rbp
        movq    %rsp,   %rbp
```

```
15          pushq   %rbx
16          pushq   %rdi
17          pushq   %rsi
18          pushq   %r12
19          pushq   %r13
20          pushq   %r14
21          pushq   %r15
22          sub     $56,    %rsp
23
24  # #t0 = 3
25          movq    $3, -64(%rbp)
26
27  # a = #t0
28          movq    -64(%rbp),  %rdx
29          movq    %rdx,   -72(%rbp)
30
31  # i
32
33  # #t1 = 0
34          movq    $0, -88(%rbp)
35
36  # b = #t1
37          movq    -88(%rbp),  %rdx
38          movq    %rdx,   -96(%rbp)
39
40  # #t2= 0
41          movq    $0,-104(%rbp)
42
43  # i = #t2
44          movq    -104(%rbp),%rdx
45          movq    %rdx,-80(%rbp)
46
47  # Goto .L1
48          jmp     .L1
49  .L0:
50
51  # i += 1
52          movq    -80(%rbp),  %rdx
53          addq    $1, %rdx
54          movq    %rdx,-80(%rbp)
```

```
55 .L1:
56
57 # #t5= i<a
58         movq    -80(%rbp),  %rdx
59         movq    -72(%rbp),  %rcx
60         cmp     %rdx,   %rcx
61         jg      1f
62         movq    $0, %rdx
63         jmp     2f
64 1:
65         movq    $1, %rdx
66 2:
67         movq    %rdx,   -112(%rbp)
68
69 # Ifz #t6 Goto .L2
70         movq    -112(%rbp), %rdx
71         cmp     $0, %rdx
72         je      .L2
73
74 # b = i
75         movq    -80(%rbp),  %rdx
76         movq    %rdx,   -96(%rbp)
77
78 # Goto .L0
79         jmp     .L0
80 .L2:
81
82 # end main
83         movq    $60,    %rax
84         xor     %rdi,   %rdi
85         syscall
86
87 # #t7 = "__main__"
88         leaq    .LC2(%rip), %rdx
89         movq    %rdx,   -64(%rbp)
90
91 # #t8 = __name__ == #t7
92         movq    (%rbp), %rdx
93         movq    -64(%rbp),  %rcx
94         movq    %rdx,   %rdi
```

```
95          movq    %rcx,   %rsi
96          call    strcmp@PLT
97          movsx   %eax,   %rax
98          cmp     $0, %rax
99          jne     1f
100         movq    $1, %rdx
101         jmp     2f
102 1:
103         movq    $0, %rdx
104 2:
105         movq    %rdx,   -80(%rbp)
106
107 # Ifz #t8 Goto .L3
108         movq    -80(%rbp),  %rdx
109         cmp     $0, %rdx
110         je      .L3
111
112 # aligning stack
113         pushq   $0
114
115 # funccall main
116         pushq   %rax
117         pushq   %rcx
118         pushq   %rdx
119         pushq   %r8
120         pushq   %r9
121         pushq   %r10
122         pushq   %r11
123
124 # call main 0
125         call    main
126         add     $0, %rsp
127         popq    %r11
128         popq    %r10
129         popq    %r9
130         popq    %r8
131         popq    %rdx
132         popq    %rcx
133         popq    %rax
134         add     $8, %rsp
```

```
135
136 # Goto .L4
137         jmp     .L4
138 .L3:
139 .L4:
```

8.3 Lists

Since list size cannot be pre-determined, especially when it is a parameter in a function, we have pointers for lists. List size, then becomes 8 bytes - the size of a pointer. We first call `malloc@PLT` and allocate the memory for list and initialize the elements of the list one by one. As lists are mutable, this approach is useful to modify individual elements of list. The first element of the list is used to store the length of the list. So if the length of the list is `n`, we actually allocate `(n+1)*8` bytes of memory.

```
1 def main():
2     a : list[int] = [2, 3, 4]
3     print(a[0])
```

```
1         .section    .rodata
2 .LC0:
3         .string "%ld\n"
4 .LC1:
5         .string "%s\n"
6 .LC2:
7         .string "__main__"
8         .globl  main
9         .text
10
11 # beginfunc main
12 main:
13         pushq   %rbp
14         movq    %rsp,   %rbp
15         pushq   %rbx
16         pushq   %rdi
17         pushq   %rsi
18         pushq   %r12
19         pushq   %r13
```

```
20          pushq   %r14
21          pushq   %r15
22          sub     $72,    %rsp
23
24  # #t0 = 2
25          movq    $2, -64(%rbp)
26
27  # #t1 = 3
28          movq    $3, -72(%rbp)
29
30  # #t2 = 4
31          movq    $4, -80(%rbp)
32
33  # #t3 = 32
34          movq    $32,    -88(%rbp)
35
36  # pushparam #t3
37          movq    -88(%rbp),  %rdi
38          call    malloc@PLT
39
40  # #t4 = %rax
41          movq    %rax,   -96(%rbp)
42
43  # len of list  = #t3
44          movq    -96(%rbp),  %rdx
45          addq    $0, %rdx
46          movq    -88(%rbp),  %rax
47          movq    %rax,   (%rdx)
48
49  # *(#t4 + 8) = #t0
50          movq    -96(%rbp),  %rdx
51          addq    $8, %rdx
52          movq    -64(%rbp),  %rax
53          movq    %rax,   (%rdx)
54
55  # *(#t4 + 16) = #t1
56          movq    -96(%rbp),  %rdx
57          addq    $16,    %rdx
58          movq    -72(%rbp),  %rax
59          movq    %rax,   (%rdx)
```

```
60
61  # *(#t4 + 24) = #t2
62          movq    -96(%rbp),  %rdx
63          addq    $24,    %rdx
64          movq    -80(%rbp),  %rax
65          movq    %rax,   (%rdx)
66
67  # a = #t4
68          movq    -96(%rbp),  %rdx
69          movq    %rdx,   -104(%rbp)
70
71  # #t5 = 0
72          movq    $0, -112(%rbp)
73
74  # #t6 = (#t5 + 1) * 8
75          movq    -112(%rbp), %rdx
76          addq    $1, %rdx
77          imulq   $8, %rdx
78          movq    %rdx,   -120(%rbp)
79
80  # #t7 = *(a + #t6)
81          movq    -104(%rbp), %rdx
82          addq    -120(%rbp), %rdx
83          movq    (%rdx), %rcx
84          movq    %rcx,   -128(%rbp)
85
86  # print(#t7)
87          movq    -128(%rbp), %rax
88          movq    %rax,   %rsi
89          leaq    .LC0(%rip), %rax
90          movq    %rax,   %rdi
91          movq    $0, %rax
92          call    printf@PLT
93
94  # end main
95          movq    $60,    %rax
96          xor     %rdi,   %rdi
97          syscall
98
99  # #t8 = "__main__"
```

```
100         leaq    .LC2(%rip), %rdx
101         movq    %rdx,   -64(%rbp)
102
103 # #t9 = __name__ == #t8
104         movq    (%rbp), %rdx
105         movq    -64(%rbp),  %rcx
106         movq    %rdx,   %rdi
107         movq    %rcx,   %rsi
108         call    strcmp@PLT
109         movsx   %eax,   %rax
110         cmp     $0, %rax
111         jne     1f
112         movq    $1, %rdx
113         jmp     2f
114 1:
115         movq    $0, %rdx
116 2:
117         movq    %rdx,   -80(%rbp)
118
119 # Ifz #t9 Goto .L0
120         movq    -80(%rbp),  %rdx
121         cmp     $0, %rdx
122         je      .L0
123
124 # aligning stack
125         pushq   $0
126
127 # funccall main
128         pushq   %rax
129         pushq   %rcx
130         pushq   %rdx
131         pushq   %r8
132         pushq   %r9
133         pushq   %r10
134         pushq   %r11
135
136 # call main 0
137         call    main
138         add     $0, %rsp
139         popq    %r11
```

```
140         popq    %r10
141         popq    %r9
142         popq    %r8
143         popq    %rdx
144         popq    %rcx
145         popq    %rax
146         add     $8, %rsp
147
148 # Goto .L1
149         jmp     .L1
150 .L0:
151 .L1:
```

8.4 Objects

Similar to list, to construct objects, we allocate the size of object in heap using `malloc@PLT`, when constructor function is called. We then assign the values of object attributes based on offsets stored in symbol table.

```
1 class Student:
2     def __init__(self, name : str, age : int):
3         self.name : str = name
4         self.age : int = age
5
6 def main():
7     s : Student = Student("Shril", 19)
8     s.age = 20
```

```
1         .section    .rodata
2 .LC0:
3         .string "%ld\n"
4 .LC1:
5         .string "%s\n"
6 .LC2:
7         .string "Shril"
8 .LC3:
9         .string "__main__"
10        .globl  main
11        .text
```

```
12
13 # beginfunc Student.__init__
14 Student.__init__:
15         pushq   %rbp
16         movq    %rsp,   %rbp
17         pushq   %rbx
18         pushq   %rdi
19         pushq   %rsi
20         pushq   %r12
21         pushq   %r13
22         pushq   %r14
23         pushq   %r15
24         sub     $8, %rsp
25
26 # *(self + 0) = name
27         movq    16(%rbp),   %rdx
28         addq    $0, %rdx
29         movq    24(%rbp),   %rcx
30         movq    %rcx,   (%rdx)
31
32 # *(self + 8) = age
33         movq    16(%rbp),   %rdx
34         addq    $8, %rdx
35         movq    32(%rbp),   %rcx
36         movq    %rcx,   (%rdx)
37
38 # endfunc__init__
39         jmp     __init__.Student_.end
40 __init__.Student_.end:
41         add     $8, %rsp
42         popq    %r15
43         popq    %r14
44         popq    %r13
45         popq    %r12
46         popq    %rsi
47         popq    %rdi
48         popq    %rbx
49         popq    %rbp
50         ret
51
```

```
52 # beginfunc main
53 main:
54         pushq   %rbp
55         movq    %rsp,   %rbp
56         pushq   %rbx
57         pushq   %rdi
58         pushq   %rsi
59         pushq   %r12
60         pushq   %r13
61         pushq   %r14
62         pushq   %r15
63         sub     $56,    %rsp
64
65 # #t0 = "Shril"
66         leaq    .LC2(%rip), %rdx
67         movq    %rdx,   -64(%rbp)
68
69 # #t1 = 19
70         movq    $19,    -72(%rbp)
71
72 # #t2 = 16
73         movq    $16,    -80(%rbp)
74
75 # call malloc 16
76         movq    -80(%rbp),  %rdi
77         call    malloc@PLT
78
79 # #t3 = %rax
80         movq    %rax,   -88(%rbp)
81
82 # funccall Student
83         pushq   %rax
84         pushq   %rcx
85         pushq   %rdx
86         pushq   %r8
87         pushq   %r9
88         pushq   %r10
89         pushq   %r11
90
91 # pushparam #t1
```

```
92          movq    -72(%rbp),  %rbx
93          pushq   %rbx
94
95  # pushparam #t0
96          movq    -64(%rbp),  %rbx
97          pushq   %rbx
98
99  # pushparam #t3
100         movq    -88(%rbp),  %rbx
101         pushq   %rbx
102
103 # call Student.__init__ 3
104         call Student.__init__
105         add     $24,    %rsp
106         popq    %r11
107         popq    %r10
108         popq    %r9
109         popq    %r8
110         popq    %rdx
111         popq    %rcx
112         popq    %rax
113
114 # s = #t3
115         movq    -88(%rbp),  %rdx
116         movq    %rdx,   -96(%rbp)
117
118 # #t4 = offset(Student,age)
119         movq    $8, %rdx
120         movq    %rdx,   -104(%rbp)
121
122 # #t5 = 20
123         movq    $20,    -112(%rbp)
124
125 # *(s + #t4) = #t5
126         movq    -96(%rbp),  %rdx
127         addq    -104(%rbp), %rdx
128         movq    -112(%rbp), %rcx
129         movq    %rcx,   (%rdx)
130
131 # end main
```

```
132         movq    $60,    %rax
133         xor     %rdi,   %rdi
134         syscall
135
136 # #t6 = "__main__"
137         leaq    .LC3(%rip), %rdx
138         movq    %rdx,   -64(%rbp)
139
140 # #t7 = __name__ == #t6
141         movq    (%rbp), %rdx
142         movq    -64(%rbp),  %rcx
143         movq    %rdx,   %rdi
144         movq    %rcx,   %rsi
145         call    strcmp@PLT
146         movsx   %eax,   %rax
147         cmp     $0, %rax
148         jne     1f
149         movq    $1, %rdx
150         jmp     2f
151 1:
152         movq    $0, %rdx
153 2:
154         movq    %rdx,   -80(%rbp)
155
156 # Ifz #t7 Goto .L0
157         movq    -80(%rbp),  %rdx
158         cmp     $0, %rdx
159         je      .L0
160
161 # aligning stack
162         pushq   $0
163
164 # funccall main
165         pushq   %rax
166         pushq   %rcx
167         pushq   %rdx
168         pushq   %r8
169         pushq   %r9
170         pushq   %r10
171         pushq   %r11
```

```
172
173 # call main 0
174         call    main
175         add     $0, %rsp
176         popq    %r11
177         popq    %r10
178         popq    %r9
179         popq    %r8
180         popq    %rdx
181         popq    %rcx
182         popq    %rax
183         add     $8, %rsp
184
185 # Goto .L1
186         jmp     .L1
187 .L0:
188 .L1:
```

8.5 Typecasting

```
1 def add(a : int, b : bool) -> bool:
2     c : int = a + b
3     return c
4
5 def main():
6     res : int = add(True, 3)
```

```
1         .section    .rodata
2 .LC0:
3         .string "%ld\n"
4 .LC1:
5         .string "%s\n"
6 .LC2:
7         .string "__main__"
8         .globl  main
9         .text
10
11 # beginfunc add
```

```
12 add:
13         pushq   %rbp
14         movq    %rsp,   %rbp
15         pushq   %rbx
16         pushq   %rdi
17         pushq   %rsi
18         pushq   %r12
19         pushq   %r13
20         pushq   %r14
21         pushq   %r15
22         sub     $40,    %rsp
23
24 # #t0 = cast_to_int b
25         movq    24(%rbp),   %rdx
26         movq    %rdx,   -64(%rbp)
27
28 # #t1 = a + #t0
29         movq    16(%rbp),   %rdx
30         addq    -64(%rbp),  %rdx
31         movq    %rdx,   -72(%rbp)
32
33 # c = #t1
34         movq    -72(%rbp),  %rdx
35         movq    %rdx,   -80(%rbp)
36
37 # #t2 = cast_to_bool c
38         movq    -80(%rbp),  %rdx
39         cmp $0, %rdx
40         jne     1f
41         movq    $0, %rdx
42         jmp 2f
43 1:
44         movq    $1, %rdx
45 2:
46         movq    %rdx,   -88(%rbp)
47
48 # %rax = #t2
49         movq    -88(%rbp),  %rax
50
51 # endfunc
```

```
52          jmp     add_.end
53  add_.end:
54          add     $40,    %rsp
55          popq    %r15
56          popq    %r14
57          popq    %r13
58          popq    %r12
59          popq    %rsi
60          popq    %rdi
61          popq    %rbx
62          popq    %rbp
63          ret
64
65  # beginfunc main
66  main:
67          pushq   %rbp
68          movq    %rsp,   %rbp
69          pushq   %rbx
70          pushq   %rdi
71          pushq   %rsi
72          pushq   %r12
73          pushq   %r13
74          pushq   %r14
75          pushq   %r15
76          sub     $56,    %rsp
77
78  # #t3 = True
79          movq    $1, -64(%rbp)
80
81  # #t4 = 3
82          movq    $3, -72(%rbp)
83
84  # #t5 = cast_to_int #t3
85          movq    -64(%rbp), %rdx
86          movq    %rdx,   -80(%rbp)
87
88  # #t6 = cast_to_bool #t4
89          movq    -72(%rbp), %rdx
90          cmp $0, %rdx
91          jne     1f
```

```
92          movq    $0, %rdx
93          jmp 2f
94  1:
95          movq    $1, %rdx
96  2:
97          movq    %rdx,   -88(%rbp)
98
99  # aligning stack
100         pushq   $0
101
102 # funccall add
103         pushq   %rax
104         pushq   %rcx
105         pushq   %rdx
106         pushq   %r8
107         pushq   %r9
108         pushq   %r10
109         pushq   %r11
110
111 # pushparam #t6
112         movq    -88(%rbp),  %rbx
113         pushq   %rbx
114
115 # pushparam #t5
116         movq    -80(%rbp),  %rbx
117         pushq   %rbx
118
119 # call add 2
120         call    add
121         add     $16,    %rsp
122
123 # #t7 = %rax
124         mov     %rax,   -96(%rbp)
125         popq    %r11
126         popq    %r10
127         popq    %r9
128         popq    %r8
129         popq    %rdx
130         popq    %rcx
131         popq    %rax
```

```
132         add     $8, %rsp
133
134 # #t8 = cast_to_int #t7
135         movq    -96(%rbp),  %rdx
136         movq    %rdx,   -104(%rbp)
137
138 # res = #t8
139         movq    -104(%rbp), %rdx
140         movq    %rdx,   -112(%rbp)
141
142 # end main
143         movq    $60,    %rax
144         xor     %rdi,   %rdi
145         syscall
146
147 # #t9 = "__main__"
148         leaq    .LC2(%rip), %rdx
149         movq    %rdx,   -64(%rbp)
150
151 # #t10 = __name__ == #t9
152         movq    (%rbp), %rdx
153         movq    -64(%rbp),  %rcx
154         movq    %rdx,   %rdi
155         movq    %rcx,   %rsi
156         call    strcmp@PLT
157         movsx   %eax,   %rax
158         cmp     $0, %rax
159         jne     1f
160         movq    $1, %rdx
161         jmp     2f
162 1:
163         movq    $0, %rdx
164 2:
165         movq    %rdx,   -80(%rbp)
166
167 # Ifz #t10 Goto .L0
168         movq    -80(%rbp),  %rdx
169         cmp     $0, %rdx
170         je      .L0
171
```

```
172 # aligning stack
173         pushq   $0
174
175 # funccall main
176         pushq   %rax
177         pushq   %rcx
178         pushq   %rdx
179         pushq   %r8
180         pushq   %r9
181         pushq   %r10
182         pushq   %r11
183
184 # call main 0
185         call    main
186         add     $0, %rsp
187         popq    %r11
188         popq    %r10
189         popq    %r9
190         popq    %r8
191         popq    %rdx
192         popq    %rcx
193         popq    %rax
194         add     $8, %rsp
195
196 # Goto .L1
197         jmp     .L1
198 .L0:
199 .L1:
```

9 Contribution

- Sawan - 33.3%
- Shrilakshmi - 33.3%
- Yashas - 33.3%